Epochs

Jon A. Solworth*

87-807

April 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*This work was supported in part under ONR grant number N00014-86-K-0215 and under NSF grant number DCR-8503610.

Epochs

Jon A. Solworth[1]

Department of Computer Science
Upson Hall
Cornell University
Ithaca, N. Y. 14853

(607) 255-7166
Solworth@cornell

Abstract

To date, the implementation of message passing languages have required the communications variables (sometimes called ports) either to be limited to the number of physical communications registers in the machine, or to be mapped to memory. Neither solution is satisfactory. Limiting the number of variables decreases modularity and efficiency of parallel programs. Mapping variables to memory increases the cost of communications and the granularity of parallelism. We present here a new programming language construct called *epochs*.

Epochs are a scoping mechanism within which the programmer can declare communications variables, which are live only during the scope of that epoch. To limit the range of time a register has to be allocated for a communications variable, the compiler ensures that all processors enter an epoch simultaneously. The programming style engendered fits somewhere between the SIMD data parallel and MIMD process spawning models.

We describe an implementation for epochs including an efficient synchronization mechanism, means of statically binding registers to communications variables and a method of fusing epochs to reduce synchronization overhead.

1. Introduction

An important class of parallel processors is that of *message passing* computers, which perform interprocessor communication and synchronization by exchanging messages. Sending a message takes place in two steps: the message is routed on a communications medium, and then stored in a location

[1]This work was supported in part under ONR grant number N00014-86-K-0215 and under NSF grant number DCR-8503610.

which is readable by the receiving processor. In a high-level language these locations are abstracted into a name space of *communications variables* (sometimes called *ports*).

The communications variables in a program must be mapped to physical locations. If communications variables are mapped to the receiver's memory, there can exist as many communications variables as there are variable names in the program. While elegant, this method has two drawbacks:

(1) Data stored in memory requires more time to access, and

(2) Messages may be requested before they arrive.

The asynchronous nature of the message arrival specified in (2) requires some mechanism for synchronizing senders and receivers of messages. This is the well known producer-consumer operating system problem (see [PeS83]). Several hardware implementations of the standard software solutions have been proposed. On the Denelcor HEP [Smi81] each location contains a valid bit which is turned on when the message arrives. A read of a location not yet valid is repeated until the message arrives. Thus, the HEP busy waits when the receiver is ready before the message arrives. An alternative method called I-structures, proposed by Arvind [ArT80], does not require busy waiting. However, I-structures require significant hardware expense to implement. In either case, the use of memory-based communications variables adds a significant overhead to the cost of message passing architectures. We estimate that for one architecture, Microflow [SoN85], the overhead of buffering a message in memory would be a factor of four over the cost of sending the message, not including the additional cost of synchronization. The overhead results not only in increased computation time, but in reduced effectiveness of fine-grain parallelism since the theoretical performance improvement from parallelization must exceed the overhead.

Communications variables can be mapped to registers instead of memory. Registers are not only significantly faster than memory cells, but dedicated hardware, such as on the Cray I's registers, can block the receiving process until the message arrives [Rus78]. When there are more communications variables than registers, this mechanism requires the overloading of registers —several different communications variables must be mapped to the same register. Since the processor that reads the value is normally not the one that wrote it, both sending and receiving processors must agree on the mapping of communications variables to registers. Moreover, to preserve the performance advantages of such a scheme we shall insist that this mapping be static.

A simple method of overloading the registers is to have one communications variable per node for each processor which can write to it. This is feasible for systems with small number of processors, such as the Alliant [All85], or for systems supporting a small number of logical connections such as the INMOS Transputer [Whi85]. However, in computers containing thousands of processors, this technique would be expensive, since the number of registers on the processor chip is limited, and since off-chip storage would incur extra cost per access. Moreover, this method requires that the sender produces messages in the same order that the receiver receives them —this has performance implications which we shall discuss latter.

An alternative method is for the compiler to map variables to different registers according to the requirements of the algorithm. We have coded about a dozen parallel algorithms in the context of the Microflow Project including FFT, quicksort, bitonic sorting, B-trees, breadth-first search and L-U decomposition. All the algorithms we have examined require, at most, the number of communications variables to be proportional to the logarithm of the number of processors [HNS87]. This seems to indicate that the number of communication variables per algorithm is quite small on average; however, a typical applications program would contain many algorithms each with its own set of communications variables. As is shown later, the liveness properties of these variables differs from those of regular variables, and hence an unbounded number of registers could be required. Given this finding, are there methods of mapping communications variables to registers which are both effective and efficient? We present a mechanism to do just this in the rest of the paper.

2. Assumptions

The proposed mechanism will exploit several assumptions we shall make about parallel programs.

(1) Parallelism is derived from the parallel execution of a single program, and hence the total program is available at compile time.

(2) Parallelism is often the result of executing the same code on different data.

Property (2) often results in fine-grain parallelism (especially for non-numeric computation), hence it is necessary to reduce the overheads of invoking this parallelism. The primary way of doing this is by statically analyzing the program and allocating resources. Property (1) makes that analysis possible.

3. The problem

Consider the case of different processors executing a single code and communicating via messages. In what follows, the processor numbered i ($processor_i$) communicates with $processor_j$ by sending messages to a register-mapped communications variable on $processor_j$. We shall examine the consequences of asynchronous execution of the processors to the communications variable mapping.

One way to synchronize the sender with receiver is to have synchronous message passing, i.e. a message is only sent when the receiver is ready to consume it. This is the method used by CSP [Hoa78]. Unfortunately, synchronous message passing both requires bi-directional communication and prohibits the pipelining of messages on stochastic networks. Bidirectional communication increases memory bandwidth requirements and doubles latency. However, a far worse effect is the prohibition of message pipelining. Since a routing network will have at least O(log(P)) diameter, the lack of message pipelining imposes a logarithmic latency on random message addresses. Hence, we will consider only asynchronous message passing.

To maximize efficiency of register utilization, it is desirable that $processor_i$ sends a message only when $processor_j$ is ready to receive that message. Unfortunately, $processor_i$ has only local state information and hence cannot determine when $processor_j$ would be ready to receive the message. So $processor_j$ must allocate a register for that communications variable the entire period during which it could receive a message from $processor_i$. During this period the communications variable is said to be *live*. Since processors operate independently, the only constraint with which to limit this period are the data dependencies. This is illustrated in figure 1. We have found that compile-time analysis of data dependencies is unlikely to yield a mapping of the communications variables to a succinct set of registers.

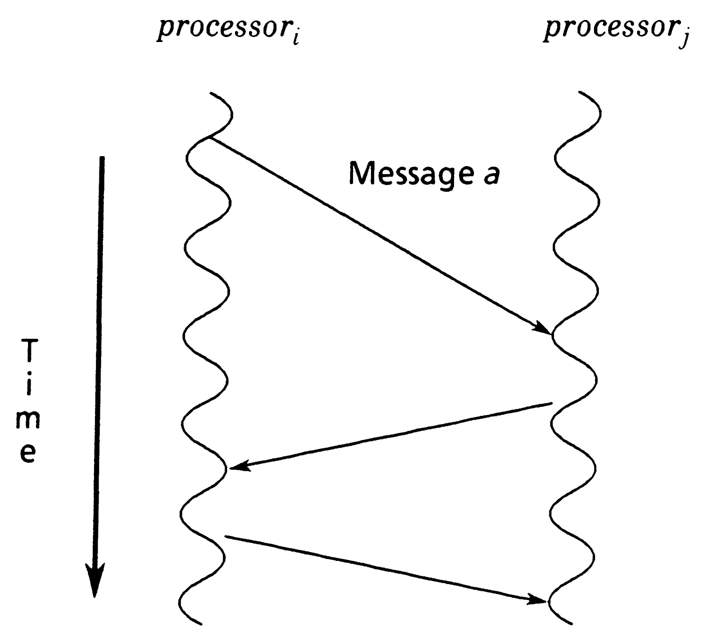

Figure 1 — Time (t) during which $processor_j$ can receive message a from $processor_i$.

Another consequence of this asynchrony is that communications variables which are assigned to registers do not follow normal scoping mechanisms. For example, assume x is an ordinary variable declared in procedure p and p calls procedure q. A compiler would assign x to a register $reg.x$ and, immediately before the call to q would generate code to save $reg.x$ in the memory location assigned to x. This is called *spilling* the value of the register [AhU77]. It is the synchronous execution of code which allows x to be spilled. If, instead of being an ordinary variable, x were a communications variable, then spilling could not occur without (expensive) synchronization with the sending process since the receiving process could not know whether a value for x would arrive when q is executing. Hence, communications variables are bound to registers for the entire time that they are live.

This has important consequences for compilation as well as register allocation. Compilation cannot be performed for procedures individually, but must be done for the sequence of procedure calls. With non-recursive procedures there are a bounded number of such sequences. However, recursive procedures present special problems.

To improve the quality of register mapping we must limit the range of time during which the message could arrive at $processor_j$. This can be done by increasing the level of synchrony in the machine. In the limit, an SIMD strategy can be employed (since execution is derived from a single code). However, our definition of message passing does not require the messages to be sent (or to arrive) in the same order that they are consumed. Hence, multiple registers must be allocated for message passing – in fact it might not be possible to bound this number at compile time. So SIMD execution does not solve the register allocation problem. Moreover, SIMD computations are not always efficient. A preferable strategy is to allow processor execution to be no more tightly bound than the sends and receives allow.

We define *skew* to be the degree of asynchrony of different processors executing the same code. In general, too small a skew results in idling processor resources, and too large a skew results in potentially unbounded number of registers. We would like to tune the skew to allow maximum execution speed while limiting register use to the number of registers in the machine. We shall say that a program has a natural (minimum) skew based on the patterns of sends and receives.

One approach is to allow the compiler, given the natural skew inherent in the program, to maximize speed by increasing skew until all registers are use. Different communications variables can be mapped to the same register only if the messages are guaranteed to arrive at the receiver in a specific order or if they are mutually exclusive in time. Our examination of several algorithms show that compile-time analysis would be unlikely to achieve significant reduction in the number of registers required.

An alternative approach is to allow the programmer to insert explicit synchronization code to ensure that communications variable use will be temporally separated. This solution requires the programmer to explicitly allocate and release communications registers, even when programming in a high-level language. This increases the programmer burden by forcing him to deal with low-level issues. As discussed in [HNS87] this also reduces the modularity or the code.

4. Model of parallel programming

Our model of parallel programming fits somewhere in between the data parallel model of Connection Machine [HiS86] and the dynamic spawning of processes as in the Ultracomputer [GGK84]. As in the data parallel model, we begin with a single program which runs on all the processors. This means that all the parallelism is available at the start of execution and as the execution proceeds some elements get "masked out".

However, the execution model is not SIMD. Processors which are not used on one branch of the computation are available for use on another. The result is that the set of processors can be recursively subdivided to work in parallel, allowing greater asynchrony (and hopefully greater efficiency) than SIMD models.

To implement this model, a program runs independently at each processing node. Picture the run-time stack for a processor as representing the execution of the program. The stack grows and shrinks vertically. Interprocessor communication takes place horizontally between processors at nearly the same stack configuration. Figure 2 shows the stack configuration of $processor_i$ sending a message to $processor_j$. The solid part of the stack represents the point at which $processor_j$ must have reached. The dotted lines represent procedures and scopes called by $processor_j$ during which time it is still legal for it to receive a message. In succeeding sections, we shall make these statements more precise.

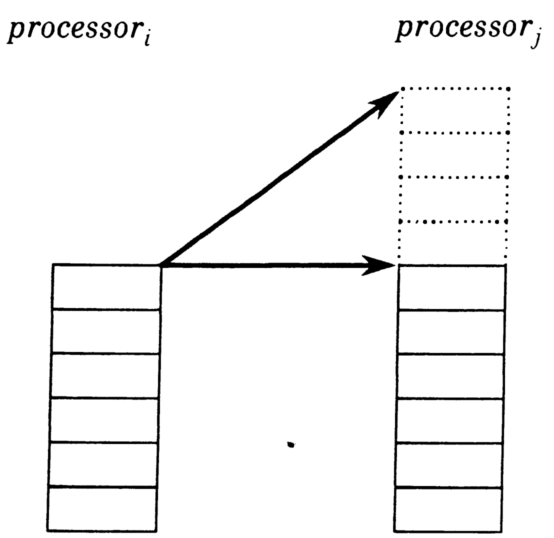

Figure 2 – Stack configuration of processors i and j when $processor_j$ is receiving a message from $processor_i$

5. Epochs

The method we propose for managing communications variables is to have periods of time during which communications variables could be live called *epochs*. An epoch spans multiple processors. As a programming language construct, epochs both declare communications variables and define their scope. Communications variables must be referenced only within the epoch in which they are lexically scoped. If $processor_i$ sends a message to $processor_j$ via communications variable x, then both $processor_i$ and $processor_j$ must be executing within the epoch where x is declared when x is sent, and $processor_j$ must be executing within the epoch when the message is received. Therefore, the only communications variables which are live at a given point of execution are those within the current (dynamically opened) epochs. Two further requirements are that a processor exits an epoch only after all the messages that will ever be sent to it during that epoch have been received, and that messages are not sent to processors that do not enter the epoch.

Epochs limit skew to a range within the dynamic execution of an epoch. We shall latter show how the compiler can merge epoch synchronizations to reduce overheads. To enable these optimizations, it is not permitted to use epoch entrance as an explicit synchronization point.

An example of epoch use is shown in Figure 3. Epoch A has two communications variables (x and y), and contains a call to procedure p, which also contains epoch C. When p is called from epoch A and epoch C is entered, there are 4 communications variables which are live – Epoch A variables x and y, and Epoch C variables x and y.

```
int
main()
        epoch A do
              com int x;
              com float y;

              p(z);
              < some code with sends and receives of x and y >
        end;

        epoch B do
              com int x;
              com int z;

              < some code with sends and receives of x and z >
        end;
end;

int
p(a)
        int a;

        epoch C do
              com int x;
              com int q;

              <sends and receives of x and q>
        end;

end;
```

Figure 3 - an epoch example

5.1. Notation

We describe the notation used in programs by an annotated BNF.

A program is composed of variable declarations and one or more *threads*.

program	←	*variable_decl* program | *thread_decl* program
thread_decl	←	*compute_thread_decl server_thread_decl**

A thread is a single code executed independently on each of the P processors in the system and can be thought of as P processes. (Processing is customized only because each processor knows its own processor number). Although a program is composed of possibly many threads, one is distinguished as the computation thread — the others are all known as server threads. The computation thread is the main focus of control, while a server thread, as the name indicates, provides a service to the computation thread.

compute_thread_decl	←	**thread** *thread_name* **do** *inside_compute_thread* **end**;
inside_compute_thread	←	*procedure_decl inside_compute_thread* | *variable_decl inside_compute_thread* | ϵ
server_thread_decl	←	**thread** *thread_name* **do** *inside_server_thread* **end**;
inside_server_thread	←	*epoch_decl inside_server_thread* | *procedure_decl inside_server_thread* | *variable_decl inside_server_thread* | ϵ

In the computation thread, procedures and variables are declared. Program execution begins at a distinguished procedure called *main* in the computation thread. A server thread consists of a number of epochs, procedures, and variables. At any time during the execution of the program, only one outer epoch per server thread can be active. A computational epoch can send to a server epoch only if the server epoch is in the outermost scope of the server thread. A server thread can send to the computational thread which instantiates it.

epoch_decl ← **epoch** *epoch_name* **do**
*cv_decl**
*epoch_instantiate**
*statement**
end

where *cv_decl* is a communications variable declaration, and *statement** contains sends and receives corresponding to the communications declared withing the epoch scope.

cv_decl ← **com** *type var*;

declares *var* as a communications variable of data type *type*.

epoch_instantiate ← **instantiate** *epoch_name*;

Declares that the current epoch will send values to a server epoch called *epoch_name* (a server epoch is an epoch which is directly inside a server thread scope). Hence, *epoch_name*'s communication variables are available as targets of sends. Moreover, the entrance of the computation epoch also creates the server epoch in a different thread.

send(*value*, *var*, *epoch_name*, *proc*) sends a *value* to an *epoch_name* at *proc*. *Var* is a communications variable declared in the epoch to which the value is sent.

receive(*var*);

Although different semantics are possible, we follow in our example the semantics of receive used in *MFL*[3] [HNS87]. Associated with every communications variable is a queue. The head of that queue is removed and assigned to variable. If the queue is empty, the next operation on the variable will block until the data arrives at the queue and is removed.

Finally, there is a keyword **proc** which specifies the number of the processor (from $0...P-1$) which is executing the code.

5.2. Epoch use

Now that the notation has been given, we wish to show an example of epoch use in a program. Our example is a parallel breadth-first search.

Figure 4(a) shows the main thread to compute a breadth-first search (*bfs*) of a graph. The outer (synchronized) loop iterates through each level of the breadth-first search. The inner loops iterate through the list of nodes which are adjacent to nodes in the current set on that processor and computes the set of nodes to search for the next level of the *bfs*. To ensure that a node shows up on the visited set at most once, access to a given node is always through a particular processor (on the server thread *node_enqueue*). The search starts with a *seed* node.

```
thread main do
    bfs(seed)

        epoch round do
            com boolean done;
            com ptr queue;
            instantiate node_enqueue;
            instantiate boolean_tree;

            done := false;

            if VertexToProc(seed) = proc then
                send(seed, elmt, node_enqueue, VertexToProc(seed));
                send(ENDMARK, elmt, node_enqueue, proc);
            end;

            while not done do
                receive(queue);
                local_done := true;
                forall v ∈ queue do
                    forall a ∈ adjacency(v) do
                        send(a, elmt, node_enqueue, VertexToProc(a))
                        local_done := false;
                    end;
                end;

                – synchronize end-of-round
                send(local_done, term, boolean_tree, parent(proc));
                receive(done);
                send(ENDMARK, elmt, node_enqueue, proc);
            end;
        end;
    end;
end;
```

Figure 4(a) Main computation thread for breadth-first search

The main thread contains a single procedure called *bfs*. Within that procedure, an epoch is declared that will make use of two server epochs (*node_enqueue* and *boolean_tree*). The procedure begins by initializing each processor's queue: exactly one will contain the seed, all others will be empty.

The function of the inner and outer loop of *bfs* has already been described. Now the servers will be described.

```
thread node do
        epoch node_enqueue do
                com ptr elmt;
                Local next_queue;

                do

                        next_queue := nil;

                        receive(elmt);
                        repeat until elmt=ENDMARK do
                                if not elmt->visited then
                                        next_queue := append(next_queue, elmt);
                                        elmt->visited := true;
                                        process(elmt);
                                end;
                                receive(elmt);
                        end;

                        send(next_queue, queue, round, proc);
                end;
        end;
end;
```

Figure 4(b) node server thread (ensures that node is put on queue at most once)

Node_enqueue (Figure 4(b)) builds a queue of elements that are adjacent to elements in the current round and that have not been encountered before in the search. Each processor considers only those elements which are physically located at the processor's node. The queue built by the processor in the *node_enqueue* thread will be used by the thread in the next round.

```
thread bt do
    epoch boolean_tree do
        com boolean term;
        com boolean result;
        boolean partial_or;

        receive(term);        – receive from computation thread
        partial_or := term;

        – receive from sons in boolean_tree thread
        forall s ∈ sons do
            receive(term);
            partial_or := partial_or or term;
        end;

        if proc=root then
            result := partial_or;
        else
            send(partial_or, term, boolean_tree, parent(proc));
            receive(result);
        end;

        send(result, done, round, proc);
        forall s ∈ sons(proc) do
            send(result, result, boolean_tree, s);
        end;
    end;
end;
```

Figure 4(c) boolean tree server thread (returns the *or* of all values).

The second server thread, *boolean_tree* (Figure 4(c)), implements a or-tree which returns true if any of its leaves are true, and also synchronizes the completion signal for the next round.

5.3. Programming-in-the-large

Now that we have given a small but detailed example of epochs, we would like to describe how epochs could be combined to build large parallel programs.

```
thread computation do
      main()
            input();                 – read in the data structure
            g := graph_create();     – create a graph
            bfs(g);                  – breadth-first search creates list
            sort(list);              – sort list
      end;

      bfs() ... end;

      process(node)
            epoch linear do
                  ...
                  instantiate add_linear;
                  ...
            end;
      end;
end

–     node_enqueue and sort_server must operate as disjoint times
thread node do
      epoch node_enqueue do ... end

      epoch sort_server do ... end;
end

thread bt do
      ...
end;

thread make_linear do

      epoch parallel_to_linear do
            ...
      end;
end;
```

Figure 5 – Programming-in-the-large

This program in figure 5 contains four series procedures at the main level to read in the data, construct the graph, produce a linear sequence from the breadth-first search and to sort that sequence. The communications variables for each of those four procedures are guaranteed to be live at disjoint times. Hence, the same set of communications registers can be used for each of the four procedures.

Also of interest is that the thread *node* contains two epochs at its topmost level. However, since those epochs will operate at disjoint time periods they can share one processor context.

6. Implementing epochs

In this section we show basic properties of epochs and show how to implement them.

6.1. Epoch match

In order to send a message to a communications variable declared in an epoch we need to ensure that both sender and receiver are executing within that epoch. There are two issues: the first is to define what it means for two processors to be in the same epoch, and the second is how to synchronize two processors that will enter the same epoch. The solution of these two issues form what we call *epoch match*. We will define the property of two processors executing in the same epoch in terms of an operational semantics on an abstract run-time stack.

Without loss of generality, we can treat loops as recursive procedures by the obvious transformation. Hence, we shall only consider procedures and conditionals as altering the control flow of the program.

A procedure definition or epoch defines a *scope*. Let N be the number of scopes occurring statically in the text of the program. Each scope is labeled with an integer between 1 and N, with the *main* procedure (the first one called) being given the label 1.

We will call procedure calls and epochs *objects of interest*. We number the objects of interest within a scope in the order that they appear in the text of the scope, and ignoring for the purposes of numbering objects which are interior to the scope of epochs. We denote the *jth* object of interest in *ith* scope as $elmt_{i,j}$.

That completes the description of the static structure of the program. We model the run-time behavior of the program by an abstract stack. Every time a new scope is entered via $elmt_{i,j}$ the pair $<i,j>$ is pushed onto the stack. When a scope is exited, the top element is removed from the stack.

We shall say that $processor_i$ and $processor_j$ are executing in the same epoch iff some initial segment (starting at the bottom) of the abstract stacks are equal, and at least one of the <i,j> pairs in that initial segment represents an epoch.

It follows from this definition that two processors which are executing in the same scope are executing within a single textual occurrence of the epoch.

Moreover, each processor dynamically reached that epoch by the same sequence of procedure calls and loop iterations.

Theorem Each epoch on a given processor occurs at most once.

proof If an abstract stack $stack = \langle\langle i_1, j_1\rangle, \langle i_2, j_2\rangle, \cdots \langle i_S, j_S\rangle\rangle$ (where the Sth element is the top of the stack) then let $Lex(stack) = \langle j_1, j_2, \cdots j_S\rangle$. We define a relation on stacks, $s_1 \geq_s s_2$ which is true iff Lex(s_1) is lexically greater than or equal to Lex(s_2). If we can show that for all successor states, the relationship $\geq_s$ holds, then we have shown that each epoch can be entered on a given processor at most once.

Let stack[1..k] represent the k bottommost elements of the stack. We wish to consider each successor stack *succ*. If succ[1..k-1] = stack[1..k-1] and $stack_{k,0} = succ_{k,0}$, we need to show that $stack_{k,1} \leq succ_{k,1}$. But how could $stack_{k,1} \leq succ_{k,1}$ fail to hold? only if control was transferred to an earlier point in the scope. However, this is only possible with loops, which we have transformed out of the program.

6.2. Epoch synchronization

An epoch which is spread across multiple processors must be synchronized in order to ensure the receiving processor is ready. It is also necessary to verify that the epochs which are being synchronized are the same. The definition of epoch match can be implemented directly but this is likely to be expensive since it means constructing an abstract stack, and comparing it with other processors' stack.

A second issue is whether to match epochs pairwise (when $processor_i$ wants to send to $processor_j$), or to match epochs for all processors simultaneously. Clearly, skew arguments favor pairwise match, but synchronization costs would in practice be high because of the large number of synchronizations per epoch. Alternatively, all of the processors could enter the epoch simultaneously. Synchronization on a PRAM [Sni85] would require $O(log(P))$ time, where P is the number of processors. Simultaneous synchronization appears desirable, but we need to know what is the set of processors which will enter an epoch. This is an undecidable problem at compile time since epochs can be on branches of conditional statements.

We will first describe the algorithm used by the compiler to automatically insert synchronization code. It is convenient to assume that we compile in a path specific knowledge, that is we compile each non-recursive sequence of procedure calls as a single entity.

A program dynamically constructs a mapping from the abstract stack states to the integers. A variable r at each processor holds the current integer representation of the abstract stack. (The value r changes each time a new scope is entered or exited). In fact, we construct a stack of r values called the $r-stack$, which is the same size as the abstract stack. The chief difference is that the r value at a given level of the stack is a representation of the corresponding abstract stack from the bottom to the level of the r

value. The following mappings are also defined:

parent(r)	The value below r on the $r-stack$ or 0 if $r-stack$ is a singleton.
new_r(curr_r, scope)	The new value of r when scope is entered with $r = curr\_r$.
$reach[r]$	number of processors which have reached stack configuration r.
$\overline{reach}[r]$	number of processors which will never enter stack configuration r.

Note that for an epoch to be entered:

$$(reach(s) + \overline{reach}[s]) = reach[parent(s)] \; s = r, parent(r), \cdots$$

Since the reach counter synchronization is needed only for epoch synchronization, a compiler can remove (or coalesce) many reach computations.

We describe how these functions are implemented. *New_r* is implemented as a global server which builds a map of <curr_r, scope> pairs to integers. If <curr_r, scope> has not been seen by the server, a value of r never before issued is returned and the map is augmented by <<curr_r, scope>, r>. Otherwise, the mapped value is provided. The implementation of this server (and several other algorithms) is described in [SXH]. $Reach[r]$ is computed by each processor calling an increment server for reach right before the new scope. $\overline{Reach}[r]$ is computed by calling the increment server on each execution path which will never reach the corresponding scope. This code is inserted by the compiler, and is only possible because there are no **goto**'s in the code.

Note that the server ensures that the *parent*(r) returns the same value on every processor on which value r is part of the $r-stack$ (and is undefined on all other processors).

This mechanism will work effectively within our context only if the number of active r values is quite small. But this is bound by the number of processors, and in practice will be even smaller.

Lemma R values are equal iff abstract stacks are equal.

proof The pushing and poping of values on the $r-stack$ exactly follows that on the abstract stack. It follows that a unique $r-value$ is constructed for each new scope by the definition of new_r. Hence every abstract stack on a processor is represented by a unique r. To show that the the same $r-value$ is use for all equal abstract stack reguardless of the processors we use simple induction. Base case is trivial since every processor begins with abstract stack $= \ll 1,1 \gg$ and $r = 0$. Induction step follows trivially from the construction of new_r.

Lemma *reach* and $\overline{reach}$ synchronize same epochs.

proof We prove by induction that the property holds. The base case $(r = 0)$ is trivial since execution on all processors begins with and $reach[0] = P$ (and $\overline{reach}[0] = 0$). The induction is on the children of r. Let s be a child of r. No more processors can enter s then entered r since s can only be entered from r. Assume that a processor enters s but does not enter r. Then, since s is the innermost scope containing r, a processor could only fail to enter r because it took an alternative execution path and $\overline{reach}[s]$ would be incremented. Otherwise, $reach[s]$ will be incremented and execution will wait until all processors which want to enter s rendezvous here. So all elements which eventually enter r will increment either $reach[s]$ or $\overline{reach}[s]$

Theorem Epoch match implements synchronization of the same epoch on different processors.

proof Follows from above two lemmas.

6.3. Sufficient Conditions

In this section we prove that a bounded number of registers are needed for communications if there are no cycles in the epoch-proc graph which we will define. Hence, given sufficient number of registers, the communications variables can be mapped to registers at compile time. The proof assumes that procedures are not passed as parameters.

Let us define an epoch-proc graph as a directed graph, with a node in the graph for each procedure and each epoch in the program. There will be an edge of type 1 from an $epoch_i$ to a $procedure_j$ if $epoch_i$ encloses a call to $procedure_j$. There is an type 2 edge from $procedure_i$ to $epoch_j$ if $epoch_j$ occurs within $procedure_i$. Finally, there is an type 3 edge from $procedure_i$ to

$procedure_j$ if $procedure_j$ is called by $procedure_i$.

Theorem There is a compile-time bound on the number of registers needed for communications variables if there are no cycles in the graph containing an epoch.

proof Each epoch declares a finite number of communications variables and there are only a finite number of epochs. Therefore, the only way to attain an unbounded number of "live" communications variables is for an epoch to be reentered without having been exited. The scoping mechanism of all constructs other than procedures (since there are no **goto**'s) ensures that these constructs cannot lead to an unbounded number of live communications variables.

To complete the proof we show that if there are an unbounded number of registers then there must be a cycle containing an epoch in the epoch-proc graph of the form:

$$procedure_i \rightarrow epoch_i' \rightarrow \cdots procedure_i$$

If there is an unbounded number of communications variables then at least one particular epoch must be entered an unbounded number of times. Call that epoch $epoch_i'$. Let $procedure_i$ be the procedure that encloses $epoch_i'$. Clearly $epoch_i'$ cannot be reentered unless $procedure_i$ is recalled. Hence, from within $epoch_i'$ there must be a call to a procedure which will eventually calls $procedure_i$. Hence, the epoch-proc map will have a cycle of the above form consisting of the following sequence of edge types: one type 1 edge, one type 2 edge, zero or more type 3 edges culminating at $procedure_i$.

6.4. Epoch Fusion

Epochs within the same thread either occur in series or are nested. Two epochs are said to be in series if it is not possible for a single processor to be in the two epochs simultaneously. Two epochs are said to be nested if it is possible for a single processor to be in them simultaneously. Epochs in series reduce the number of communications registers needed by ensuring that the communications variable operate during disjoint times. Nested epochs, on the other hand, do not decrease the number of communications variables outstanding. This is because at the inner most nesting level all of the communications variables are live.

Since the register requirements of communications variables for nested epochs sum (at the inner-most epoch), and because the parents of the same epoch are always the same we examine the possibility of fusing epochs. Epoch fusion is a technique for reducing the number of epoch synchronizations. In the case of nested epochs, epoch fusion does not increase maximum number of communications registers required.

Epoch fusion is defined on pairs of epochs. Let e_1, e_2 be epochs such that on the path we are compiling e_1 dynamically encloses e_2 without enclosing any intervening epoch. We wish to coalesce the synchronization for e_1 and e_2 at e_1 and allocate the communications registers for e_2 when e_1 is entered.

Epoch fusion both reduces the overhead for creating epochs and enables greater skew (and hence higher processor utilization). Moreover, if epoch fusion is applied only to nested epochs there is no increase in the worse case number of communications variables.

Theorem Coalescing epochs does not change the semantics of epoch operation.

proof Note that the communication of e_1's scope does not change at all. We are therefore left with examining the consequences of e_2's communications. Let $processor_i$ send a message to $processor_j$ within scope e_2. Clearly, $processor_i$ will be in e_2 when the message is sent (since the name cannot appear outside the static scope of e_2). If $processor_j$ is executing in e_1 when the message arrives, the message will wait at the allocated communications register. $Processor_j$ will be in e_2 when the message is received, and must eventually enter the epoch since it would be illegal otherwise to send the message to $processor_j$. $Processor_j$ will not receive messages after it leaves e_2 since an epoch cannot exit until all of its messages have been accepted. Of course, some processors will enter the fused epoch which would not have entered e_2. But since these processors will neither send nor be sent to within this epoch, their execution is unaffected.

6.5. Communications variable overflow

There are two cases in which we will not be able to map communications variables to registers. The first is when the number of registers required, although bounded, exceeds the number of physical registers in the machine. The second is when there is a cycle in the epoch-proc graph containing a epoch. In this case there could be unbounded number of communications variables.

For both of these cases, it is possible to use an auxiliary server thread which will simulated communications registers using memory [HNS87]. Thus, these cases can be handled, albeit with a degradation in speed for those variables which are memory mapped.

7. Summary of epoch requirements

(1) Communications variables are used only in the epoch in which they have been declared,

(2) A processor within an epoch does not send a message to a processor that never enters the epoch, and

(3) A processor receives all of the messages sent to it for an epoch before exiting that epoch.

8. Conclusions and future work

We have described a new programming structure called an epoch which focuses the time that a communications variable is active between the time that an epoch enters and exits. The programming style engendered fits somewhere between the MIMD process spawning and SIMD data parallel styles. Epochs also enable an arbitrary number of communications variables to be mapped to a succinct number of registers.

We have described an implementation for epochs including an efficient synchronization mechanism, a means of statically binding registers to communications variables and a method of fusing epochs to reduce synchronization overhead.

We are currently implementing epochs as part of the MFL^3 compiler for the Microflow computer. This will enable efficient and modular implementation of message passing on applications. We will also be able to answer the following questions:

(1) How much time is spent waiting for epoch synchronization over non-epoch based systems?

(2) How much do epochs reduce the number of communications registers used?

(3) What is the cost of synchronization and how much is it reduced by epoch fusion?

For the long term, we are looking at higher level languages which will compile into message based systems.

Acknowledgements

Patrick Xavier read through an early (almost unreadable) version of this paper and made many suggestions to improve the presentation and clarity of ideas. Alex Nicolau provided a sounding board for the ideas while they were taking shape. Marc Snir suggested improvements which increased the elegance of threads.

References

[AhU77] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.

[All85] Alliant, *Product Summary*, Alliant Computer Systems Corporation, Acton Mass., January, 1985.

[ArT80] Arvind and R. H. Thomas, "I-Structures: An Efficient Data Type for Functional Languages", *Laboratory of Computer Science, Tech. Memo. 178*, 1980.

[GGK84] A. Gottlieb, R. Grishman, C. P. Kruskal, D. P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer – designing an MIMD shared memory parallel processor", *Institute of Electrical and Electronics Engineers Transactions on Computers C-33*, 11 (November 1984).

[HNS87] L. Hendren, A. Nicolau, J. A. Solworth and P. Xavier, "Low Level Programming For a Massively Parallel Fine-Grain Computer: the Microflow Approach", *C. S. Dept*, February 1987.

[HiS86] W. D. Hillis and G. L. Steele, Jr., "Data Parallel Algorithms", *Communications of the Association for Computing Machinery 29*, 12 (December 1986), 1170-1183.

[Hoa78] C. A. R. Hoare, "Communicating sequential processes", *Comun. Association for Computing Machinery 21*, 8 (Aug. 1978), 666-677.

[PeS83] J. L. Peterson and A. Silberschatz, *Operating System Concepts*, Addison-Wesley, Reading, MA, 1983.

[Rus78] R. M. Russel, "The Cray-1 Computer System", *Communications of the Association for Computing Machinery 21*, 1 (January, 1978), 63-72.

[Smi81] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system", *Real Time Signal Processing IV, Proceedings of SPIE 298* (1981), 241-248, The International Society of Optical Engineering.

[Sni85] M. Snir, "On Parallel Searching", *Society for Industrial and Applied Mathematics J. on Computing 14*, 3 (1985), 688-708.

[SoN85] J. A. Solworth and A. Nicolau, "Microflow: A fine-grain parallel processing approach", *C. S. Dept*, Nov. 1985.

[SXH] J. A. Solworth, P. Xavier and L. Hendren, "Microflow Algorithms", *C. S. Dept*, . In preparation.

[Whi85] C. Whitby-Strevens, "The Transputer", *12th International Symposium on Computer Architecture*, Boston, Mass., June, 1985, 292-300.